Big Data Analysis Exploiting Genome Similarity
Qualifying Exam

Kristal Curtis

Dec. 13, 2012

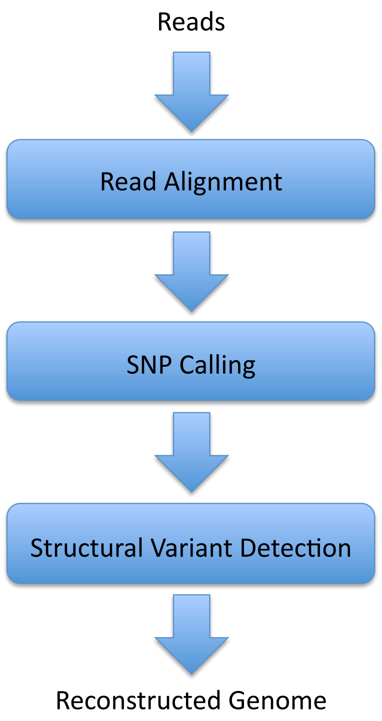


Figure 2: Variant calling pipeline, at a high level.

Abstract

1 Introduction

2 Improving Alignment with SNAP

3 The Similarity Problem

After implementing the basic SNAP algorithm, we explored how various properties of its search parameters and input data affect its speed and accuracy. Through this process, we discovered that the main obstacle to both speed and accuracy in alignment is the *similar regions* in the genome. In what follows, we discuss our observations of how similar regions impact SNAP's performance. Then, we describe a method for identifying the similar regions, as well as some insights into their characteristics. We also discuss how we plan to exploit genome similarity throughout the pipeline to improve speed and accuracy of variant calling.

3.1 Observations

One of our goals following the development of SNAP was to determine the impact of its various parameters on its speed and accuracy. By accuracy, I mean to encompass both % aligned and % error. Thus, we performed a parameter sweep. We observed that the only parameter that significantly impacted SNAP's performance was h_{max}, which is the maximum number of genome locations to which a seed could match and still be considered by SNAP. Seeds occurring at more positions in the genome than h_{max} are ignored by SNAP, to avoid spending undue time attempting to match reads that are unlikely to be matched uniquely.

Figure 2 shows the results of this parameter sweep. As we increase h_{max}, the alignment accuracy improves, *i.e.,* we are able to align more reads while making fewer errors. However, this comes at a significant cost to throughput. The interesting thing about this result is that while you might think that a read that contains popular seeds cannot be uniquely mapped, what we actually observe is that these reads that map to lots of locations actually can be mapped uniquely if we check enough locations, hence the improvement to % aligned.

We discovered that this is due to the repetitive nature of the genome. Exact duplicates are not as confounding, since if the problem were just that there are too many exact duplicates, increasing h_{max} would not result in any increase in % aligned, since it would be impossible to find a unique mapping location no matter how many locations SNAP considers. The

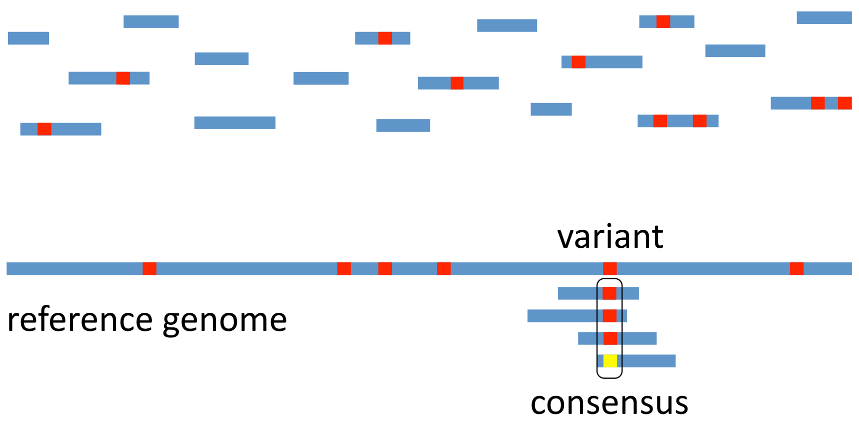


Figure 1: Variant calling overview.

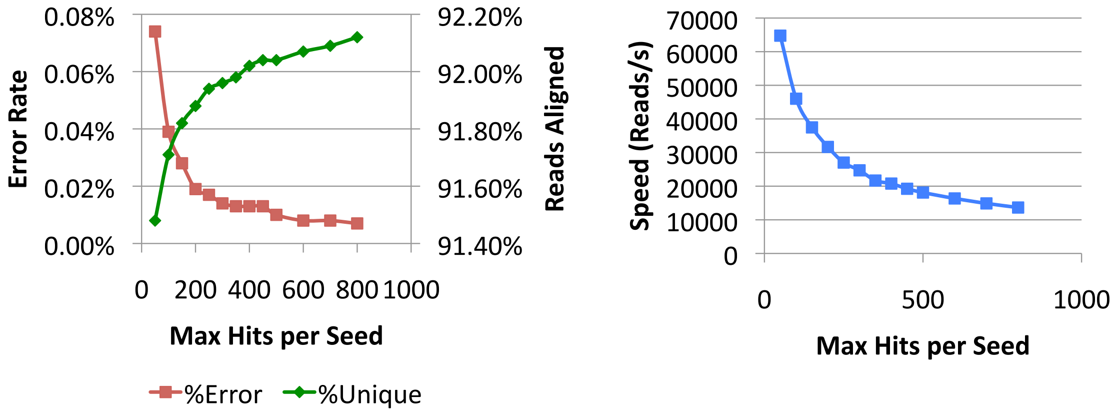


Figure 3: Accuracy improves as we increase h_{max}, the number of locations we check per seed, but throughput suffers.

difficulty here is caused by near duplicates, which we refer to as *similar regions*. The presence of near duplicates means that in many cases, it actually is possible to find a unique alignment for a read that contains popular seeds, due to slight differences between the similar regions. Thus, there is a benefit to checking many locations per read, even though it is expensive.

To confirm our intuition that similar regions are presenting a significant obstacle to alignment performance, we did a simple experiment. We aligned one million reads, simulated from the whole genome, with SNAP. Then, for the reads which SNAP aligned incorrectly, we checked whether the read belonged to a cluster. How the clusters were located will be described in Section 3.2. Our findings were surprising; out of the 227 errors that SNAP made, 195 (86%) were in clusters. For reference, only 8% of the positions in hg19 belong to a cluster. Therefore, the clusters, though making up a small fraction of the genome, make a huge impact on alignment performance. Thus, it is essential to improve the handling of these reads in order to improve alignment performance.

3.2 Identifying Similar Regions

The straightforward approach to find clusters would be to create a graph where the nodes are the substrings in the genome and there is an edge between any two nodes that are sufficiently similar. Then, the clusters correspond to the connected components of this graph. However, since there are over 3 billion substrings of length 100, the length of the short reads we are using, both building this graph and then the subsequent step to find the connected components are very expensive. Therefore, we chose to approximate this approach by using the well-known *union-find algorithm*. In union-find, each substring starts in its own cluster. Then, whenever two substrings are sufficiently similar, we merge the two clusters. The output of this process is similar to the connected components of the graph identified by the straightforward approach; however, the union-find implementation is much more efficient. In order to determine which clusters to merge, we use the SNAP index to find all locations to which a substring's seeds map. Thus, we avoid doing the full $N \times N$ comparison of all the substrings in the genome.

Though the union-find algorithm is more efficient than the naïve connected components approach, it is non-trivial to implement it so that it will run in a practical amount of time on the whole genome. From the initial version that ran on chromosome 22 to the current version that runs on the full hg19, several implementation changes were needed. The main change was going from a serial implementation to a parallel version using Spark [4]. The parallel version finds clusters in each partition, and then it merges the clusters to produce clusters that are present in the genome overall (Figure 4). Once we had a parallel implementation, we had to do several tricks to make it scale, such as changing the partitioning scheme, using a feature of Spark called accumulators to avoid storing intermediate results, and changing the representation of genome locations so that they could be stored in 4-byte integers instead of 8-byte longs. These changes were mainly motivated by saving memory.

Compared to other approaches to clustering the genome (see Section 5.2), our approach has several advantages. First, because our approach is driven by the characteristics of the short reads themselves, rather than being motivated by a more biological desire to better understand the repeat characteristics of the genome, our problem is more constrained. Many of the previous approaches to clustering the genome attempt full generality, without imposing any constraints on repeat length, and allowing many only marginally related substrings to belong to the same repeat family in the interest of reducing the number of repeat families to enhance interpretability. However, this results not only in algorithms that are very complicated and computationally expensive but also in output that is difficult to utilize for short-read processing. Our more constrained problem, where we fix the length of any substrings considered to be the same as that of the short reads, is simpler to tackle and is therefore more amenable to an efficient algorithm.

The downside to our approach is that since our setting is completely unsupervised, the clusters are difficult to evaluate via an intrinsic metric (*i.e.,* one focused solely on the clusters rather than on how the clusters benefit an application in which they are used). In the situation where labeled data exists (*e.g.,* [2]), various metrics such as cluster purity that give a sense for how cohesive the clusters are may be used. However, in our setting, we lack labeled data. Some of the previous work attempted to get around this issue by comparing to the data in RepBase. However, this comparison is harder in our case, due to the different constraints of our problem. Therefore, we rely mostly on extrinsic evaluation criteria – *i.e.,* , whether our clusters improve the performance of short read analysis.

We used a merge distance of 4 based on our experiments with varying the merge distance and looking at the results on single chromosomes. With smaller distances, the clusters turn out to be too small to make much impact. With larger distances, the downstream

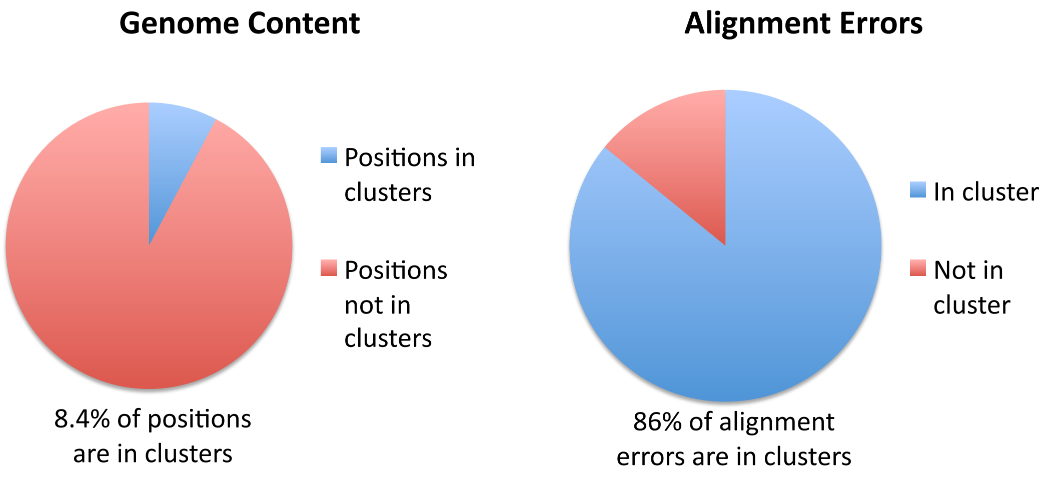


Figure 4: Despite the fact that a small fraction of the genome belongs to a cluster, a significant fraction of alignment errors occur in clusters.

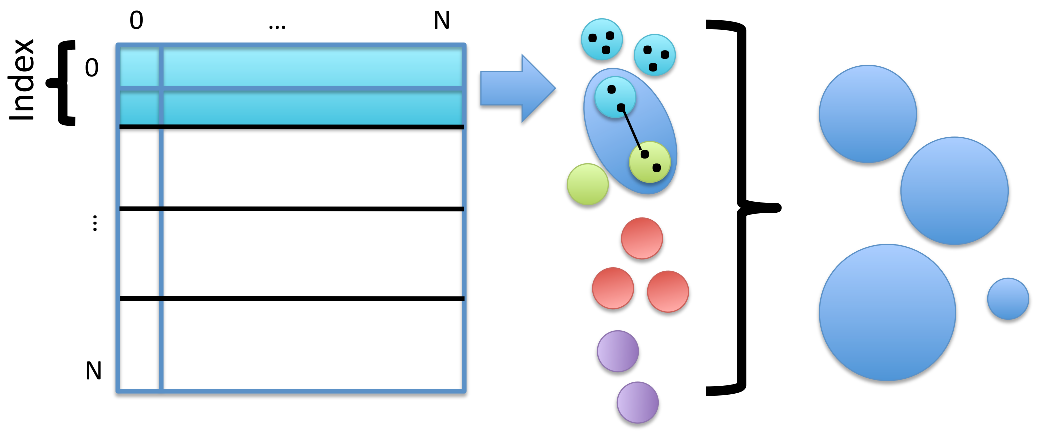


Figure 5: Union-find implementation via Spark. Parallel implementation involves partitioning the genome. We find clusters in each partition separately and then merge them to obtain clusters in the genome overall.

performance degrades too much because the clusters get too bloated. However, this choice of merge distance is impacted by the extrinsic criteria that are identified, which we will discuss in more detail in Section 4.1. The similar region finder tool applied to hg19 took 126 hours on a single node with 16 cores, or about 2016 CPU-hours. While this took a few days, it is important to note that the results could be obtained faster by scaling out onto multiple nodes with just a change to the job configuration since we used Spark. The tool used about 100 GB of memory, which is practical on modern servers. The output size is 2.8 GB, with 39,992,540 clusters, where a cluster is defined as having at least two members. The number of genome positions in these clusters is 263,230,347, or 8.4% of the 3.1B positions in hg19.

4 Vision for Exploiting Similar Regions

In Section 3, we discussed how genome similarity makes short-read processing, specifically alignment, more challenging, and we presented a method for locating the similar regions in the genome. In this section, we will discuss how to exploit the similar regions that we have identified to improve short-read analysis. We will begin by describing ongoing work that aims to improve alignment performance by using similar regions. We will also discuss plans for future work in which we will make use of the information about similar regions in subsequent stages of the pipeline.

4.1 Exploiting Similarity in Alignment

The most straightforward way to utilize the clusters we produced is to augment SNAP so that it uses information about whether a particular genome position belongs to a cluster. In the regular version of SNAP described in Section 2, alignment errors are usually due to the situation illustrated in Figure 5(a). Due to the setting of h_{max}, SNAP may consider only some of a read's candidate locations. This leads to alignment errors when SNAP chooses a location out of the ones it has considered, while the correct location was skipped.

Our idea is to extend SNAP to be aware if a read's candidate locations are in a cluster. This idea is illustrated in Figure 5(b). Instead of considering each location on its own, without regard to its cluster status, we check for each of a read's candidate locations whether that location belongs to a cluster. If it does, we automatically pull in all the members of the cluster and compare the read to each of them. This helps us avoid missing good locations where our read might align (due to h_{max} being set too low).

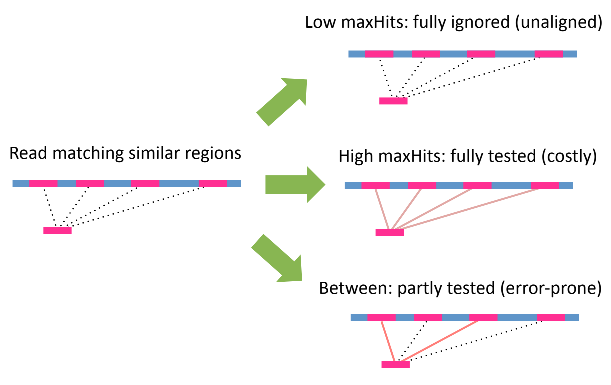
This approach is bound to improve alignment accuracy, since the effect of the clusters is that you compare against more locations and are therefore more likely to find the best location to which the read aligns. The main challenge, then, is to perform this similarity-aware alignment efficiently.

Presently, we use three techniques to improve the similarity-aware SNAP's performance. First, we create a special index that is informed by the cluster membership of the genome positions. In the normal SNAP index, we insert into the hash table a given seed and the position(s) at which it may be found in the genome. In our similarity-aware index, we modify this approach, in that if a seed found at a given position is in a cluster, instead of storing that position with the seed in the index, we store the cluster ID, which we define to be the first (*i.e.,* with the lowest index) member of the cluster. This avoids an extra step of determining, once you know that a position is in a cluster, which cluster to which a position belongs.

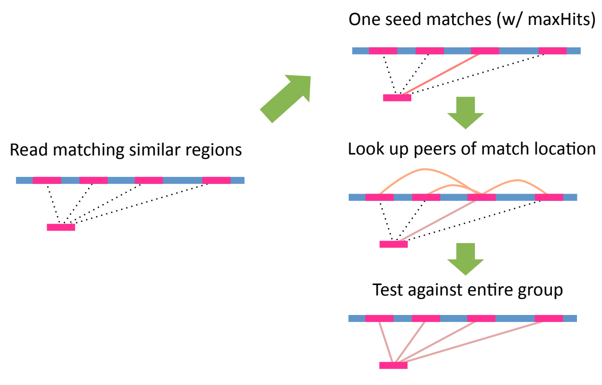
A second technique we use to reduce alignment latency is intra-cluster pruning. The idea here is that we want to be able to tolerate clusters that may be a little too inclusive, since clusters with more members result in higher accuracy (as more locations are being checked), and because as mentioned in Section 3.2, tuning the clusters from our union-find approach is not straightforward. Therefore, we introduce a preprocessing step in which for each cluster, we determine the consensus string (by simply taking the most popular base at each position), as well as each member's differences from the consensus. Then, when we decide we want to compare a read to a particular cluster, we use the triangle inequality to filter out members that we know are too distant from the read.

Third, we have developed a version of the edit distance algorithm that exploits the similarity among the cluster's members. The idea here is that the strings in a cluster are all very similar to each other, so it is wasteful to compare the read separately against all the locations. Instead, we compare only against the differences between each cluster member and the consensus string.

Another application of the clusters related to alignment is creating an *all matcher* version of SNAP, where instead of returning the single best location where a read matches (or no location, if no confident alignment is available), we return all locations to which SNAP matches within a given edit distance. All matchers tools are useful for several rea-


Low maxHits: fully ignored (unaligned)
Read matching similar regions
High maxHits: fully tested (costly)
Between: partly tested (error-prone)

(a)


One seed matches (w/ maxHits)
Read matching similar regions
Look up peers of match location
Test against entire group

(b)

Figure 6: SNAP, with and without clusters. (a) Without clusters, SNAP may miss some good locations for a read, causing errors. (b) With clusters, when SNAP finds that a read matches well against a particular location, it also checks the peers (*i.e.,* cluster members) of that location. This avoids missing locations to which the read may align better than the given location and cuts down on the error rate.

sons. First, some of the structural variant detection tools (*e.g.,* VariationHunter) rely on such information [3]. Current all matchers, such as mrFAST [1], are painfully slow. Therefore, creating a faster version of an all-match aligner would make it easier and more practical to use tools like VariationHunter. We provide the performance results of our all-matcher in Table 1. Through the use of some of the techniques presented in the context of the best-match version of SNAP (see Section 4.1), we are able to achieve a speedup over the regular SNAP all-matcher.

With both the best-match and all-match versions of SNAP, we have spent some time doing profiling to determine how to further improve the speedup obtained by using the clusters. Our main insight is that the similarity-aware versions of SNAP spend a lot more time in cache misses, each of which costs hundreds of CPU cycles, than the regular version of SNAP. These cache misses result when the similarity-aware SNAP queries the similarity map to determine whether a given position is in a cluster because the similarity map is too large to fit in the cache. Therefore, we are currently devising a strategy to alter the similarity map representation so that it will fit into the L3 cache; we are aiming to make it fit in 8 MB. We plan to use Bloom filters to represent the positions, so that we will need only 5 bits per position. We also plan to store only the cluster IDs in the similarity map, rather than all the positions that are in clusters. We expect that this will enable us to further improve the speedup obtained. However, given that SNAP's memory access patterns have been heavily optimized, the fact that the less mature versions of our similarity-aware best- and all-match algorithms are near or slightly outperforming the standard SNAP is encouraging.

4.2 Exploiting Similarity throughout the Pipeline

5 Related Work

5.1 Alignment

5.1.1 Seed-based Aligners

5.1.2 Burrows-Wheeler Transform

5.1.3 Alignment to Similar Regions

5.2 Clustering the Genome

6 Research Timeline

7 Conclusion

References

[1] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature Genetics*, 41(10):1061–1067, October 2009.

[2] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, pages 105–118, 2005.

[3] F. Hormozdiari, C. Alkan, E. E. Eichler, and S. C. Sahinalp. Combinatorial algorithms for structural

Algorithm	**Reads/s**	**Speedup**
Regular SNAP	2125	-
Sim-aware SNAP, regular index	2358	1.11
Sim-aware SNAP, sim-aware index	2920	1.37

Table 1: Using clusters in the all matcher version of SNAP results in a speedup over the regular SNAP all-matcher without clusters. Results shown were obtained aligning one million simulated reads from chromosomes 1-3.

variation detection in high-throughput sequenced genomes. *Genome Research*, 19:1270–1278, 2009.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*, April 2012.